Extracting Smart Contracts Tested and Verified in Coq

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen and Bas Spitters

Aarhus University, Concordium Blockchain Research Center

Certified Programs and Proofs 2021

COBRA
CONCORDIUM BLOCKCHAIN
RESEARCH CENTER AARHUS

AARHUS
UNIVERSITY

Motivation: smart contracts

Smart contracts (SCs):

programs in a general-purpose language running "on a blockchain"

Blockchain $\sim$ database, smart contracts $\sim$ stored procedures.

Motivation: smart contracts

Smart contracts (SCs):

programs in a general-purpose language running "on a blockchain"

Blockchain $\sim$ database, smart contracts $\sim$ stored procedures.

What is so special about them?

- Manage money: auctions, crowdfunding, multi-signature wallets, ...
- Immutable.
- Code is Law.
- Call other (possibly malicious) contracts.
- Flaws may result in huge financial losses:
  - The DAO $\sim$ \$50M — hacker attack.
  - Parity's multi-signature wallet $\sim$ \$280M — bugs in the library code.

Functional smart contract languages

- Contracts are (partial) state transition functions:

```
contract : CallCtx * Msg * State -> option (State * Action list)
```

- A *scheduler*
  - updates the state;
  - handles transfers and calls to other contracts in `Action list`.

Examples of such languages:
LIGO (Tezos), Liquidity (Dune), Scilla (Zilliqa), Sophia (Æternity), etc.

From Coq to blockchain

- Smart contract verification is crucial.
- Coq is well-suited for functional smart contracts.
- We want to do all the development in Coq.
- Use ConCert for verification.[1]

[1] DA, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A Smart Contract Certification Framework in Coq. CPP'2020

From Coq to blockchain

- Smart contract verification is crucial.
- Coq is well-suited for functional smart contracts.
- We want to do all the development in Coq.
- Use ConCert for verification.[1]

Problem: How do we get the executable code out it?

[1] DA, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A Smart Contract Certification Framework in Coq. CPP'2020

From Coq to blockchain

- Smart contract verification is crucial.
- Coq is well-suited for functional smart contracts.
- We want to do all the development in Coq.
- Use ConCert for verification.[1]

Problem: How do we get the executable code out it?

Solution: Code extraction!

[1] DA, Jakob Botsch Nielsen, and Bas Spitters. ConCert: A Smart Contract Certification Framework in Coq. CPP'2020

Code extraction

- **Translation/compilation/code generation from formal developments in proof assistants.**
- Available in Coq, Adga, Isabelle/HOL, . . .
- Common extraction target: functional programming languages.

Code extraction

- **Translation/compilation/code generation from formal developments in proof assistants.**
- Available in Coq, Adga, Isabelle/HOL, ...
- Common extraction target: functional programming languages.

Extraction in Coq

- Supports extraction to OCaml, Haskell and Scheme.
- General idea: turn computationally irrelevant bits into □ (a **box**).
- Proofs (propositions) and types appearing in terms become boxes.
- The underlying theory: Pierre Letouzey's PhD thesis.

Code extraction

- **Translation/compilation/code generation from formal developments in proof assistants.**
- Available in Coq, Adga, Isabelle/HOL, . . .
- Common extraction target: functional programming languages.

Extraction in Coq

- Supports extraction to OCaml, Haskell and Scheme.
- General idea: turn computationally irrelevant bits into □ (a **box**).
- Proofs (propositions) and types appearing in terms become boxes.
- The underlying theory: Pierre Letouzey's PhD thesis.
- ✗ Not directly suitable for targeting:
  - functional smart contract languages;
  - functional fragments of multi-paradigm languages (e.g. Rust)
- ✗ Current Coq extraction is not verified.
- ✓ MetaCoq erasure is verified!

The MetaCoq project

MetaCoq: Formalising Coq in Coq

Consists of several subprojects.

- Template Coq — adds meta-programming facilities to Coq:
  - reflects Coq's kernel;
  - quote/unquote.
- PCUIC (Predicative Cumulative Calculus of Inductive Constructions) — meta-theory of Coq.
- Safe Checker — verified reduction machine, conversion checker and type checker for PCUIC.
- **Erasure** — a verified erasure procedure.[2]

[2] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq.

MetaCoq's verified erasure

- A translation from PCUIC (the calculus of Coq) into $\lambda\Box$, untyped lambda-calculus with an additional constant $\Box$.
- Provides a proof that the evaluation of the original and the erased terms agree.

Missing bits for practical use:

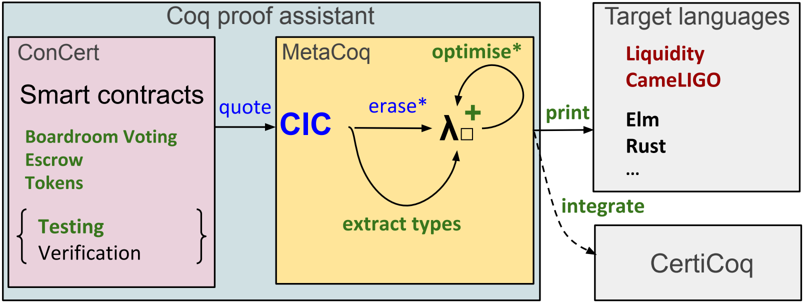
- No erasure for types and inductives.
- Requires more optimisations (e.g. removing boxes).

Contributions

- Real-world SC implementations:
  - Boardroom Voting.
  - Escrow.
  - Implementation of tokens: EIP/ERC-20, FA2.
- Testing SCs for preliminary bug discovery.
- Executable code generation through extraction.
- Verified optimisations of extracted code.
- Code extraction useful for various targets, not only smart contracts.

The pipeline

In **green** — our contributions.
Transformations marked with * — verified.



TCB: the usual of Coq + MetaCoq quote + printing + target language.

Our extraction

We extend MetaCoq:

- Extraction of types and data type definitions
  (crucial for targeting typed languages)
- A verified optimisation procedure:
  - removes unused arguments of functions;
  - removes "logical" arguments (types and proofs) of constructors.

Our extraction

We extend MetaCoq:

- Extraction of types and data type definitions
  (crucial for targeting typed languages)
- A verified optimisation procedure:
  - removes unused arguments of functions;
  - removes "logical" arguments (types and proofs) of constructors.

On top of this — pretty-printing to various targets.

Challenges of new targets

- No `Obj.magic`/`unsafeCoerce` (only available in Rust)
- Non-recursive data types only — Liquidity, CameLIGO.
- Limited support for recursion (e.g. tail recursion only, or no direct access to recursion — only through primitives) — Liquidity, CameLIGO.

Challenges of new targets

- No `Obj.magic`/`unsafeCoerce` (only available in Rust)
- Non-recursive data types only — Liquidity, CameLIGO.
- Limited support for recursion (e.g. tail recursion only, or no direct access to recursion — only through primitives) — Liquidity, CameLIGO.

Consequences:

- Some extracted code will not be well-typed.
- Remapping (cf. `Extract Inlined Constant`) is mandatory for some recursive definitions
  (also crucial for performance on a blockchain)

Deboxing

```
Definition square (xs : list nat) : list nat :=
    @map nat nat (fun x : nat ⇒ x ∗ x) xs.
```

Deboxing

```
Definition square (xs : list nat) : list nat :=
        @map nat nat (fun x : nat ⇒ x ∗ x) xs.
```

Erases to (implicit type arguments become boxes):

```
fun xs ⇒ Coq.Lists.List.map □□ (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

We want to remove the boxes:

```
fun xs ⇒ Coq.Lists.List.map (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

The same for constructors: `pair □□ a b` becomes `pair a b`

Deboxing

```
Definition square (xs : list nat) : list nat :=
    @map nat nat (fun x : nat ⇒ x ∗ x) xs.
```

Erases to (implicit type arguments become boxes):

```
fun xs ⇒ Coq.Lists.List.map □□ (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

We want to remove the boxes:

```
fun xs ⇒ Coq.Lists.List.map (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

The same for constructors: `pair □□ a b` becomes `pair a b`

- removes redundant computations;
- makes remapping e.g. `Coq.Lists.List.map` to a target language `map` easier.

Deboxing

```
Definition square (xs : list nat) : list nat :=
      @map nat nat (fun x : nat ⇒ x ∗ x) xs.
```

Erases to (implicit type arguments become boxes):

```
fun xs ⇒ Coq.Lists.List.map □□ (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

We want to remove the boxes:

```
fun xs ⇒ Coq.Lists.List.map (fun x ⇒ Coq.Init.Nat.mul x x) xs
```

The same for constructors: `pair □□ a b` becomes `pair a b`

- removes redundant computations;
- makes remapping e.g. `Coq.Lists.List.map` to a target language `map` easier.

Deboxing — a transformation that removes redundant boxes.

When (and why) is it safe to remove boxes?

- Constants: after unfolding `(fun x ⇒ t) v` $\sim$ `t`, if `x` is not free in `t`
- Deboxing is a special case: `(fun A x ⇒ t)` $\square \sim$ `(fun x ⇒ t)`.
- Constructors: boxes don't carry any useful information.
- We remove boxes from applications of constants and constructors.

The "dearging" optimisation

De-arging: removing the "dead" arguments

```
Definition foo (n m k : nat) := n.
```

Optimises to

```
Definition foo (n : nat) := n.
```

The "dearging" optimisation

De-arging: removing the "dead" arguments

```
Definition foo (n m k : nat) := n.
```

Optimises to

```
Definition foo (n : nat) := n.
```

Consists of two stages:

- Analysis: generate masks for constants and constructors (e.g. for `foo`: mask = `[f;t;t]`).
- Dearg: remove arguments using masks, adjust all usage sites.

The "dearging" optimisation

De-arging: removing the "dead" arguments

```
Definition foo (n m k : nat) := n.
```

Optimises to

```
Definition foo (n : nat) := n.
```

Consists of two stages:

- Analysis: generate masks for constants and constructors (e.g. for `foo`: mask = `[f;t;t]`).
- Dearg: remove arguments using masks, adjust all usage sites.

Note:

- if a constant or constructor is not applied "enough" — η-expand.
- With η-expansion, it might not be an optimisation.

Soundness theorems

Soundness wrt. a big-step call-by-value evaluation relation $\Sigma \vdash t \triangleright v$

Theorem (Soundness of dearging)

Let Σ, t *be a valid environment and a term of* $\lambda\Box$, η-**expanded according to provided masks**, *then*

$\Sigma \vdash t \triangleright v$ *implies* $\texttt{dearg\_env}(\Sigma) \vdash \texttt{dearg}(t) \triangleright \texttt{dearg}(v)$

Theorem (Soundness of extraction)

Let Σ *be a valid CIC environment,* C *a constant in* Σ, Σ', C *— an environment and a constant after extraction and optimisations, then*

$\Sigma \vdash_p C \triangleright Ctor$ *implies* $\Sigma' \vdash C \triangleright Ctor$

We get the η-expansion premise in a *certifying* way: quote a term, expand it, unquote back and generate a proof.

Matching the semantics

What is $\square$ in a target language?
We want it to be `() : Unit` — doesn't quite work.

Matching the semantics

What is $\square$ in a target language?
We want it to be () : `Unit` — doesn't quite work.
Semantic discrepancies between $\lambda\square$ and targets related to $\square$.

- Applied $\square$: if $t_1 \triangleright \square$ and $t_2 \triangleright v$ then $(t_1 \;\; t_2) \triangleright \square$.
  Requires unsafe features — impossible in most our targets.
  So, we have to pick () : `Unit`.
  Good news: doesn't come up often, if it does — correctness is not compromised (the target program will not be well-typed)

Matching the semantics

What is $\square$ in a target language?
We want it to be `()` : `Unit` — doesn't quite work.
Semantic discrepancies between $\lambda\square$ and targets related to $\square$.

- Applied $\square$: if $t_1 \triangleright \square$ and $t_2 \triangleright v$ then $(t_1 \;\; t_2) \triangleright \square$.
  Requires unsafe features — impossible in most our targets.
  So, we have to pick `()` : `Unit`.
  Good news: doesn't come up often, if it does — correctness is not compromised (the target program will not be well-typed)
- Pattern-matching on $\square$.
  Recent optimisation in MetaCoq removes such cases. Integrated into our development.

Matching the semantics

What is $\square$ in a target language?
We want it to be () : `Unit` — doesn't quite work.
Semantic discrepancies between $\lambda\square$ and targets related to $\square$.

- Applied $\square$: if $t_1 \triangleright \square$ and $t_2 \triangleright v$ then $(t_1 \;\; t_2) \triangleright \square$.
  Requires unsafe features — impossible in most our targets.
  So, we have to pick () : `Unit`.
  Good news: doesn't come up often, if it does — correctness is not compromised (the target program will not be well-typed)
- Pattern-matching on $\square$.
  Recent optimisation in MetaCoq removes such cases. Integrated into our development.

Guarantees (given the TCB):
extracted program type checks $\longrightarrow$ gives the same result as the original.

A counter contract

```
Inductive msg :=
  Inc (_ : Z)
| Dec (_ : Z).

Definition storage := Z.

Definition pos := {z : Z | 0 <? z}.

Program Definition inc_counter (st : storage) (inc : pos) :
  {new_st : storage | st <? new_st} := st + inc.
Next Obligation. (* proof omitted *) Qed.
...

Program Definition counter (msg : msg) (st : storage) : option storage :=
  match msg with
  | Inc i ⇒ match (bool_dec (0 <? i) true) with
            | left h ⇒ Some (inc_counter st (exist i h))
            | right _ ⇒ None
            end
  | Dec i ⇒ ...
  end.
```

A counter contract

```
Inductive msg :=
  Inc (_ : Z)
| Dec (_ : Z).

Definition storage := Z.

Definition pos := {z : Z | 0 <? z}.

Program Definition inc_counter (st : storage) (inc : pos) :
  {new_st : storage | st <? new_st} := st + inc.
Next Obligation. (* proof omitted *) Qed.
...

Program Definition counter (msg : msg) (st : storage) : option storage :=
  match msg with
  | Inc i ⇒ match (bool_dec (0 <? i) true) with
             | left h ⇒ Some (inc_counter st (exist i h))
             | right _ ⇒ None
             end
  | Dec i ⇒ ...
  end.
```

A counter contract

```
Inductive msg :=
  Inc (_ : Z)
| Dec (_ : Z).

Definition storage := Z.

Definition pos := {z : Z | 0 <? z}.

Program Definition inc_counter (st : storage) (inc : pos) :
  {new_st : storage | st <? new_st} := st + inc.
Next Obligation. (* proof omitted *) Qed.
...

Program Definition counter (msg : msg) (st : storage) : option storage :=
  match msg with
  | Inc i ⇒ match (bool_dec (0 <? i) true) with
            | left h ⇒ Some (inc_counter st (exist i h))
            | right _ ⇒ None
            end
  | Dec i ⇒ ...
  end.
```

Extracted code

```
type Msg
  = Inc Int
  | Dec Int

type alias Storage = Int

type Sig a = Exist a

type alias Pos = Sig Int

proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a

inc_counter : Storage → Pos → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage → Option Storage
counter msg st =
  case msg of
    Inc i →
      case bool_dec (lt 0 i) True of
        Left →
          Some (proj1_sig (inc_counter st (Exist i)))
        Right →
          None
    Dec i → ...
```

Listing 1: Elm

Extracted code

```
type Msg
  = Inc Int
  | Dec Int

type alias Storage = Int

type Sig a = Exist a

type alias Pos = Sig Int

proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a

inc_counter : Storage → Pos → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage → Option Storage
counter msg st =
  case msg of
    Inc i →
      case bool_dec (lt 0 i) True of
        Left →
          Some (proj1_sig (inc_counter st (Exist i)))
        Right →
          None
    Dec i → ...
```

Listing 1: Elm

Extracted code

```
type Msg
  = Inc Int
  | Dec Int

type alias Storage = Int

type Sig a = Exist a

type alias Pos = Sig Int

proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a

inc_counter : Storage → Pos → Sig Storage
inc_counter st inc = Exist (add st (proj1_sig inc))
...
counter : Msg → Storage → Option Storage
counter msg st =
  case msg of
    Inc i →
      case bool_dec (lt 0 i) True of
        Left →
          Some (proj1_sig (inc_counter st (Exist i)))
        Right →
          None
    Dec i → ...
```

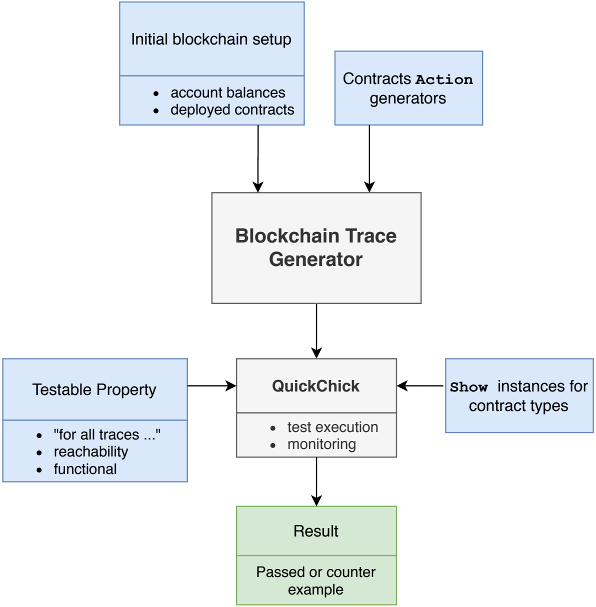
"Logical" bits are erased

Listing 1: Elm

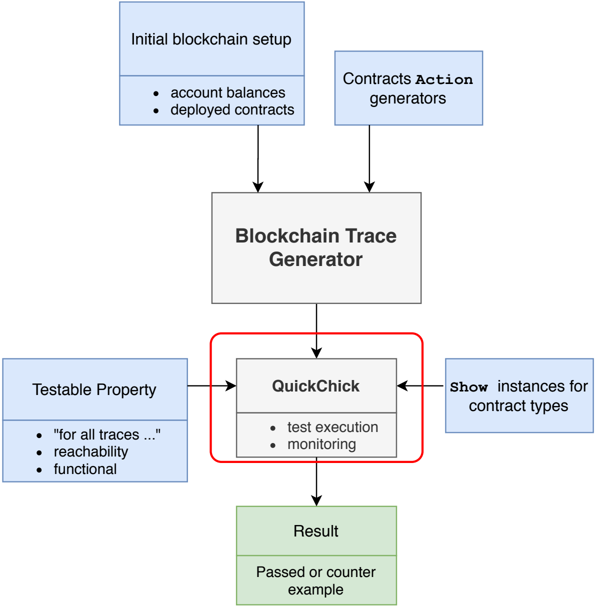
Experience with extraction

- We have extracted several SCs:
  counter, crowdfunding, prototype DSL interpreter, escrow.
- Extraction to Elm:
  list functions from Coq's stdlib, counter, `safe_head` (false elim), the Ackermann function (well-founded recursion).
- Liquidity and CameLIGO have many restrictions, requres remapping.
- Elm is closer to $\lambda\square$, extraction is more principled.
- Rust: several examples from Coq's stdlib, small examples with dependent types, graph coloring (from the CertiCoq benchmark) — WIP.

Overview of the Testing Framework

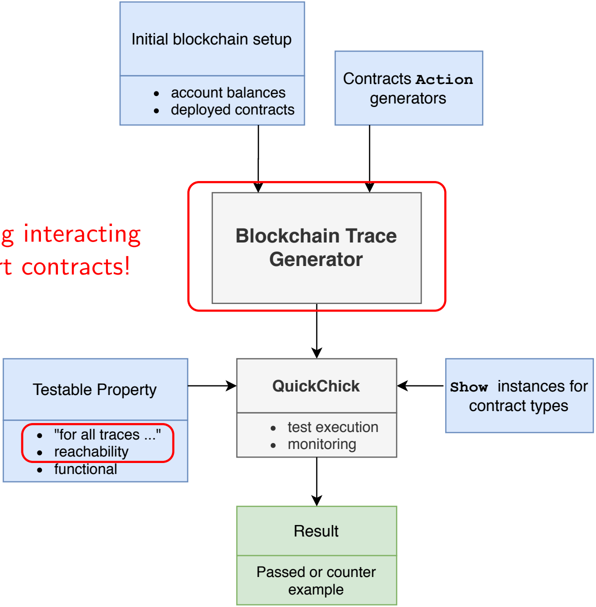


Overview of the Testing Framework



Overview of the Testing Framework

Testing interacting smart contracts!



Testing Smart Contracts

- Cost-effective, semi-automated approach to discover bugs.
- Allows for finding bugs earlier, helping the verification efforts.
- The first to support testing on execution traces.
- Case studies: Tokens (ERC20, FA2), Escrow, Congress, UniSwap.
- We have (re-)discovered many known vulnerabilities/bugs.

Testing Smart Contracts

- Cost-effective, semi-automated approach to discover bugs.
- Allows for finding bugs earlier, helping the verification efforts.
- The first to support testing on execution traces.
- Case studies: Tokens (ERC20, FA2), Escrow, Congress, UniSwap.
- We have (re-)discovered many known vulnerabilities/bugs.

Tricky bits

- Requires specialised generators defined manually (otherwise too many discards).

More smart contracts in ConCert

- Boardroom voting
  - computes a public tally from private votes;
  - tallying is based on finite field arithmetic;
  - verified, extraction — WIP.
- Escrow
  - a common contract in "decentralised finance" (DeFi);
  - tested, verified and extracted.
- Tokens
  - widely used to represent various assets;
  - tested and extracted.

Conclusions and Future work

- Practical use of the MetaCoq's erasure, thanks to our extensions.
- Verified optimisations of the extracted code.
- A step towards verified extraction framework.
- New extraction languages: Liquidity, CameLIGO, Elm and Rust.
- Smart contract testing on execution traces.
- Real-world smart contract tested and verified.
- Available as part of ConCert (https://github.com/AU-COBRA/ConCert).

Conclusions and Future work

- Practical use of the MetaCoq's erasure, thanks to our extensions.
- Verified optimisations of the extracted code.
- A step towards verified extraction framework.
- New extraction languages: Liquidity, CameLIGO, Elm and Rust.
- Smart contract testing on execution traces.
- Real-world smart contract tested and verified.
- Available as part of ConCert (https://github.com/AU-COBRA/ConCert).

Other outcomes

- Our optimisation pass integrated to CertiCoq.
- Contributions to MetaCoq.

Conclusions and Future work

- Practical use of the MetaCoq's erasure, thanks to our extensions.
- Verified optimisations of the extracted code.
- A step towards verified extraction framework.
- New extraction languages: Liquidity, CameLIGO, Elm and Rust.
- Smart contract testing on execution traces.
- Real-world smart contract tested and verified.
- Available as part of ConCert (https://github.com/AU-COBRA/ConCert).

Other outcomes

- Our optimisation pass integrated to CertiCoq.
- Contributions to MetaCoq.

What's next?

- More optimisation passes.
- Inserting type coercions, if supported by a target language.
- Change erasure to remove the "applied box" problem.

Thank you!